Some Java Concurrency Exercises

December 11, 2017

Problem 1. Fork-Join Parallelism: Longest Series

Consider the problem of finding the longest sequence of some number in an array of numbers: `longest_sequence(i,arr)` returns the longest number of consecutive `i` in `arr`. For example, if `arr` is {`2,17,17,8,17,17,17,0,17,1`} then `longest_sequence(17,arr)` is 3 and `longest_sequence(9,arr)` is `0`.

(a) In pseudocode, give a parallel fork-join algorithm for implementing `longest_sequence`. Do *not* employ a sequential cut-off: your base case should process an array range containing one element. Hint: Use this definition:

```
class Result {
  int numLeftEdge;
  int numRightEdge;
  int numLongest;
  boolean entireRange;
  Result(int l, int r, int m, boolean a) {
    numLeftEdge=l; numRightEdge=r; numLongest=m; entireRange=a;
  }
}
```

For example, `numLeftEdge` should represent the length of the sequence at the *beginning* of the range processed by a subproblem. Think carefully about how to combine results.

(b) In English, describe how you would make your answer to part (a) more efficient by using a sequential cut-off. In pseudocode, show the code you would use below this cut-off.

Problem 2. Fork-Join Parallelism: Leftmost Occurrence of Substring

Consider the problem of finding the leftmost occurrence of the sequence of characters `cseRox` in an array of characters, returning the index of the leftmost occurrence or `-1` if there is none. For example, the answer for the sequence `cseRhellocseRoxmomcseRox` is `9`.

(a) In English (though some high-level pseudocode will probably help), describe a fork-join algorithm similar in design to your solution in problem 1. Use a sequential cut-off of at least 6 (the length of `cseRox`) and explain why this significantly simplifies your solution. Notice you still must deal with the leftmost occurrence being "split" across two recursive subproblems.

(b) Give a much simpler fork-join solution to the problem that avoids the possibility of a "split" by using slightly overlapping subproblems. Assume a larger sequential cut-off, for example 100. Give your solution precisely in pseudocode. Avoid off-by-one errors.

Problem 3. Parallel Prefix and Pack

In this problem, the input is an array of strings and the output is an array of integers. The output has the length of each string in the input, but empty strings are filtered out. For example:

```
[ "", "", "cse", "rox", "", "homework", "", "7", "" ]
```

produces output:

```
[ 3, 3, 8, 1]
```

A parallel algorithm can solve this problem works by doing a parallel map to produce a bit vector, followed by a parallel prefix over the bit vector, followed by a parallel map to produce the output.

Show the intermediate steps for the algorithm described above on the example above. For each step, show the tree of recursive task objects that would be created (where a node's child is for two problems of half the size) and the fields each node needs. Do not use a sequential cut-off. Show three separate trees (for the three steps). Explain briefly what each field represents.

Note that because the input length is not a power of two, the tree will not have all its leaves at exactly the same height.

Problem 4. Another Wrong Bank Account

Note: The purpose of this problem is to show you something you should not do because it does not work.

Consider this pseudocode for a bank account supporting concurrent access:

```
class BankAccount {
  private int balance = 0;
  private Lock lk = new Lock();
  int getBalance() {
    lk.acquire();
    int ans = balance;
    lk.release();
    return ans;
  }
  void setBalance(int x) {
    lk.acquire();
    balance = x;
    lk.release();
  }
  void withdraw(int amount) {
    lk.acquire();
    int b = getBalance();
    if(amount > b) {
      lk.release();
      throw new WithdrawTooLargeException();
    }
    setBalance(b - amount);
    lk.release();
  }
}
```

The code above is wrong if locks are *not* re-entrant. Consider the *absolutely horrible idea* of "fixing" this problem by rewriting the `withdraw` method to be:

```
  void withdraw(int amount) {
    lk.acquire();
    lk.release();
    int b = getBalance();
    lk.acquire();
    if(amount > b) {
      lk.release();
      throw new WithdrawTooLargeException();
    }
    lk.release();
    setBalance(b - amount);
    lk.acquire();
    lk.release();
  }
```

(a) Explain how this approach prevents blocking forever unlike the original code.

(b) Show this approach is incorrect by giving an interleaving of two threads in which a withdrawal is forgotten.

Problem 5. Concurrent Queue with Two Stacks

Consider this Java implementation of a queue with two stacks. We do not show the entire stack implementation, but assume it is correct. Notice the stack has synchronized methods but the queue does not. The queue is incorrect in a concurrent setting.

```
class Stack<E> {
  ...
  synchronized boolean isEmpty() { ... }
  synchronized E pop() { ... }
  synchronized void push(E x) { ... }
}
```

```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue() {
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```

(a) Show the queue is incorrect by showing an interleaving that meets the following criteria:

   i. Only one thread ever performs `enqueue` operations and that thread enqueues numbers in increasing order (1, 2, 3, ...).

   ii. There is a thread that performs two `dequeue` operations such that its first `dequeue` returns a number larger than its second `dequeue`, which should never happen.

   iii. Every `dequeue` succeeds (the queue is never empty).

   Your solution can use 1 or more additional threads that perform `dequeue` operations.

(b) A simple fix would make `enqueue` and `dequeue` synchronized methods. Explain why this would never allow an `enqueue` and `dequeue` to happen at the same time.

(c) To allow an `enqueue` and a `dequeue` to operate on a queue at the same time (at least when `out` is not empty), we could try either of the approaches below for `dequeue`. For each, show an interleaving that demonstrates the approach is broken. Your interleaving should satisfy the three properties listed in part (a).

```
E dequeue() {
  synchronized(out) {
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```

```
E dequeue() {
  synchronized(in) {
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
  }
  return out.pop();
}
```

(d) Provide a solution, based on two stacks as above, that correctly allows an `enqueue` and a `dequeue` to happen at the same time, at least when `out` is not empty. Your solution should define `dequeue` and involve multiple locks.

Problem 6. Simple Concurrency with B-Trees

Note: Real databases and file systems use very fancy fine-grained synchronization for B-Trees such as "hand-over-hand locking" (which we did not discuss), but this problem considers some relatively simpler approaches.

Suppose we have a B Tree supporting operations `insert` and `lookup`. A simple way to synchronize threads accessing the tree would be to have one lock for the entire tree that both operations acquire/release.

(a) Suppose instead we have one lock per node in the tree. Each operation acquires the locks along the path to the leaf it needs and then at the end releases all these locks. Explain why this allegedly more fine-grained approach provides absolutely no benefit.

(b) Now suppose we have one readers/writer lock per node and `lookup` acquires a read lock for each node it encounters whereas `insert` acquires a write lock for each node it encounters. How does this provide more concurrent access than the approach in part (a)? Is it any better than having one readers/writer lock for the whole tree (explain)?

(c) Now suppose we modify the approach in part (b) so that `insert` acquires a write lock *only for the leaf node* (and read locks for other nodes it encounters). How would this approach increase concurrent access? When would this be *incorrect*? Explain how to fix this approach without changing the asymptotic complexity of `insert` by detecting when it is incorrect and in (only) those cases, starting the `insert` over using the approach in part (b) for that `insert`. Why would reverting to the approach in part (b) be fairly rare?

Problem 7. Concurrent/Parallel Graph Traversal

(a) In Java or pseudocode, define an *unbounded stack* with operations `push` and `pop` that can be used by threads concurrently. Make the element type generic and use a linked list for the underlying implementation. A `pop` should not raise an error. Instead it should wait until the stack is not empty. Use a condition variable. (Java detail: The `wait` method throws an exception that Java requires you catch, but it doesn't matter if you include this detail in your solution.)

(b) The method `traverseFrom` in the code below uses your answer to part (a) to apply the `doIt` method to every node reachable from some node in a graph. The order `doIt` is applied to nodes does not matter. The code uses 4 threads to try to improve performance. All you need to do is write the `run` method (probably around 10 lines) for `GraphWalker` using the other code. Requirements:

- No node should have `doIt` applied to it more than once. You can assume the `visited` field of each node is initially `false` for this purpose.
- Use the stack to hold nodes that still need to be processed. That way threads not busy completing a `doIt` call can find useful work.
- Avoid all synchronization errors. (Hint: Your code does not need any *explicit* synchronization since it can call other code that already performs synchronization.)
- Because `doIt` may be expensive, do not hold any locks while calling it.

Note that after all reachable nodes have been processed, all threads will be blocked forever waiting for the stack to become non-empty. That is fine for a homework problem. Also note that you may or may not find `isVisited` helpful.

```
class GraphNode {
  boolean visited = false;
  GraphNode[] neighbors;
  // ... other fields, constructor,
  // etc. omitted ...
  synchronized boolean isVisited() {
    return visited;
  }
  synchronized boolean wasAndSetVisited(){
    boolean ans = visited;
    visited = true;
    return ans;
  }
}
```

```
class GraphWalker extends java.lang.Thread {
  Stack<GraphNode> stk;
  GraphWalker(Stack<GraphNode> s) {
    stk = s;
  }
  void doIt(GraphNode n) {/*...*/}

  // ... FOR YOU ...

}
class Main {
  public static final int NUM_THREADS = 4;
  void traverseFrom(GraphNode root) {
    Stack<GraphNode> stk =
       new Stack<GraphNode>();
    stk.push(root);
    for(int i = 0; i < NUM_THREADS; i++)
      new GraphWalker(stk).start();
  }
}
```